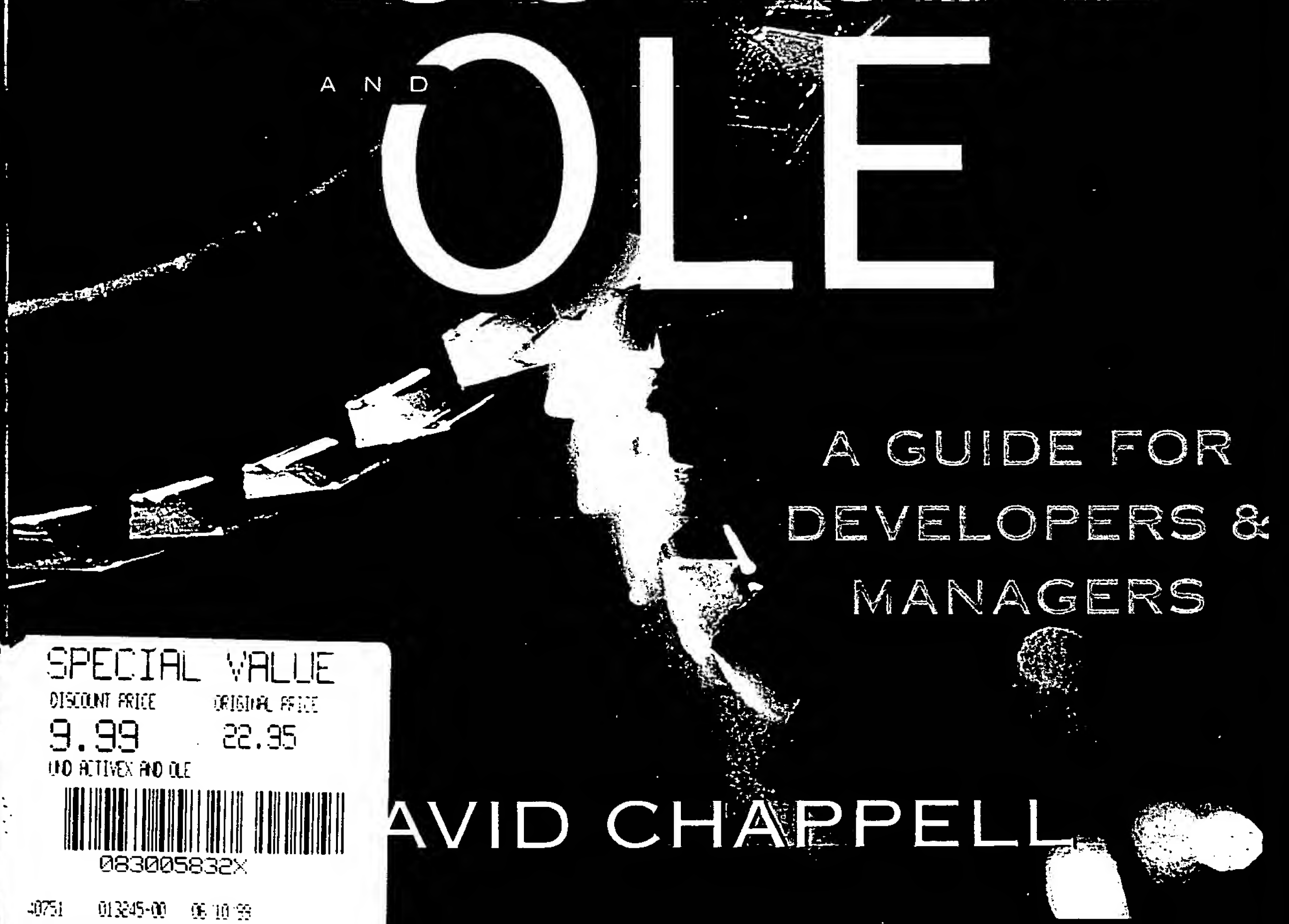
STRATEGIC TECHNOLOGY SERIES

UNDERSTANDING ActiveX™ AND OLE



A GUIDE FOR DEVELOPERS & MANAGERS

SPECIAL VALUE
DISCOUNT PRICE ORIGINAL PRICE
9.99 22.95
UND ACTIVEX AND OLE
083005832X
40751 013245-00 06 10 99

AVID CHAPPELL

Microsoft Press

Understanding ActiveX and OLE

Published by **Microsoft Press**
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data pending.

Printed and bound in the United States of America.

4 5 6 7 8 9 QMQM 1 0 9 8

Distributed in Canada by ITP Nelson, a division of Thomson Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com.

Acquisitions Editor: **David Clark**
Project Editors: **Stuart J. Stuple and Mary Renaud**
Technical Editor: **Stuart J. Stuple**
Manuscript Editor: **Mary Renaud**

An Implementor's View

To a user, the OLE technology that creates compound documents blurs the distinction between different applications, allowing a more "document-centric" view. An implementor, however, remains well aware that the container and the server are distinct applications and that each has its own duties.

An Aside: Complexity vs. Tools

The available tools make it relatively easy to support compound documents

Before launching into a more detailed discussion of OLE compound documents, there's an important point to be made: it's not as hard as it looks. True, if every application developer were required to explicitly write the code for every method in every one of the necessary interfaces, it would indeed be hard. But the available tools make it much easier. Using Microsoft's Visual C++ and Microsoft Foundation Classes (MFC), for example, the programmer need only choose specific options that these tools provide to have basic support automatically included for various compound-document features. While it's useful and interesting to know what's really going on, the level of knowledge required of an implementor seems to shrink every day—the tools do a lot of the work.

Containers

Both containers and servers must support standard interfaces

The most common examples of containers are stand-alone applications that users can start and work with, such as Word and Excel. (Remember, however, that both of these examples are also capable of acting as servers.) Building a basic OLE container is not terribly difficult—it requires supporting only two interfaces, described in "Embedding Containers," on page 180. Because the interfaces are entirely generic, a container need never know what kinds of servers are providing linked or embedded data for its documents. The container simply supports the standard container interfaces, while the server supports the standard server interfaces. Neither is aware of what kind of application the other is. In our example, for instance, Word knows only that it's interacting with some server that follows OLE's rules for servers; it doesn't know, or care, that the server is Excel.

Servers

Servers aren't quite as straightforward as containers. They support more interfaces than containers do, and some of these interfaces are fairly complex. Servers also come in different configurations, and the creator of an OLE server must decide which one is right for that application.

A local server runs as a separate process

The choices mirror the options for building a server for any COM object. One choice is to implement a server for compound documents as a local server, running as an entirely separate process. Excel takes this approach when it's acting as a server. In our example, Word and Excel run as two separate processes, yet they cooperate to present the user with one compound document.

An in-process server runs as a DLL in the container

A second choice is to implement a server for compound documents as an in-process server. Here the server can't run on its own but is instead a DLL, dependent on being loaded into a container. This is absolutely the right choice for some kinds of servers. One common (and important) example of an in-process server that uses some parts of the technology for compound documents is an ActiveX control, discussed in the next chapter.

There's a complication, however: even a local OLE server, one like Excel that runs in a separate process, needs to maintain a representative in the container's process. To understand why, look back at Figure 8-3. When the user edits the Excel spreadsheet in a separate window, the changes can also be reflected in the view of the spreadsheet that's part of the compound document. Excel itself can reflect the changes in the window it controls, but Excel doesn't control the compound-document window. That window is owned by the container, Word, and only something in the container's process can write to that window. To show the user's edits in the compound-document window, then, Excel (or any local server) must tell its representative in the container about changes to its data and then let that representative actually reflect those changes on the screen.

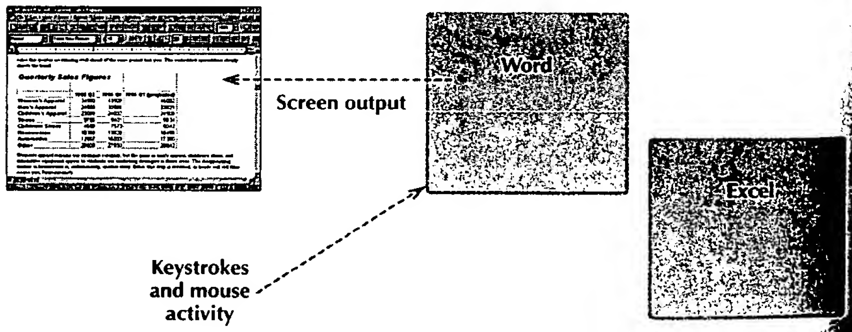
A local server relies on an in-process handler in the container

This representative, called an *in-process handler,* can take a couple of different forms. If the server is implemented entirely in a separate process, it can rely on the default handler, a standard implementation provided by Microsoft. Alternatively, if the server implements some of its functionality in a separate process and some in an in-process server linked to the container, it can provide its own handler, which is typically built on top of the default handler implementation. Whichever choice is made, a good deal of the work involved in making this Word/Excel example happen revolves around communication between a local server and its in-process surrogate.

Cooperation Between Containers and Servers

Suppose that a user opens our example compound document and edits the part of the document owned by Word. As Figure 8-4 illustrates, all user input—everything that the user types, all mouse clicks within this window, and so on—is sent directly to Word, and Word reflects the changes being made by writing directly to its part of the window. Although it's shown in the figure, it's possible that Excel isn't even running yet. (More on this later.)

Figure 8-4 ***Editing the Word part of the compound document.***

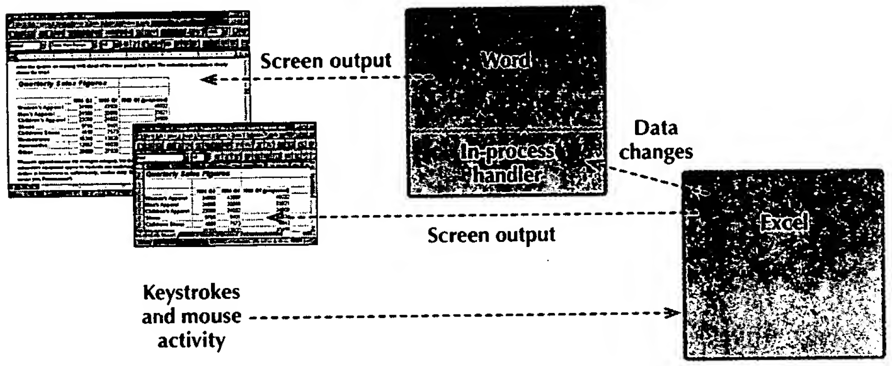


Now suppose that the user decides to edit the Excel spreadsheet in a separate window. The situation becomes a little more complicated, as shown in Figure 8-5. In this case, all of the user's input to the spreadsheet is sent directly to Excel. Excel carries out commands just as it would if the user were editing a stand-alone Excel spreadsheet rather than one that is embedded or linked. The challenge for Excel is to make the view of the spreadsheet within the compound document reflect the changes as they are made. Because Excel can't write directly to that view, it must instead inform its surrogate, the in-process handler loaded by Word, about the user's edits. The in-process handler can then work together with Word to ensure that those changes appear in the view of the spreadsheet seen by the user. (When the user chooses in-place activation rather than editing in a separate window, the situation illustrated in Figure 8-5 is only slightly different. See "In-Place Activation," page 189.)

Changes to a local server's data are sent to the in-process handler

Figure 8-5

Editing the Excel spreadsheet in its own window.



The dynamics of getting the right information to the right place at the right time and then correctly displaying this information to the user are the central problems to be solved in implementing compound documents. The remainder of this chapter is devoted to a closer look at how this is done, beginning with embedding.

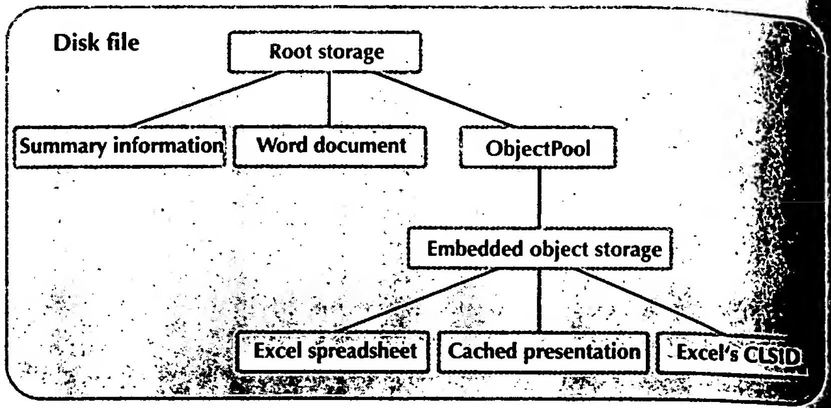
The central problem is making the user's changes appear correctly everywhere

How Embedding Works

Embedding, in which the server's data is physically stored in the same file as the container's data, is typically accomplished using Structured Storage (discussed in Chapter 5). If we assume that the

Excel spreadsheet in our example is embedded in the Word document, the compound file would resemble the somewhat simplified picture shown in Figure 8-6.

Figure 8-6 *A simplified picture of a compound file containing a Word document with an embedded Excel spreadsheet.*



Word creates a storage called ObjectPool to store embedded objects

The text of the Word document is stored in a stream just below the compound file's root storage. The document summary information, described in Chapter 5, also has its own stream below the root. Immediately below the file's root storage is a storage called ObjectPool.[3] Because many objects can be embedded in a single file, embedding even one object causes Word to create an ObjectPool storage. Each object instance that's embedded in this Word file gets its own storage below ObjectPool (labeled *Embedded object storage* in the figure). This example includes only one embedded object, the Excel spreadsheet, so only a single storage is shown below ObjectPool.

An embedded document is stored beneath its own storage object

This single storage belongs to the embedded Excel object. Below this storage, one stream contains the data for the Excel spreadsheet, and another contains Excel's CLSID. (Without the CLSID, how would the container know which application to start to work with the embedded object's data?) Several more streams exist here,

3 This is how Word stores embedded objects, but there are no hard-and-fast rules for how this must be done. Many container applications do follow a similar model, however.